A Collection of Patterns for Use Case Maps

Gunter Mussbacher and Daniel Amyot[1]
School of Information Technology and Engineering (SITE), University of Ottawa
161 Louis-Pasteur, PO Box 450, Stn. A, Ottawa (ON), Canada, K1N 6N5
damyot@site.uottawa.ca

Our gratitude extends to Jim Coplien for his efforts as our shepherd.

Abstract: Use Case Maps (UCMs) are a technique used to capture functional requirements and high level designs of complex systems composed of many features. Once you have chosen UCMs as part of your software development process, the question arises as how to most effectively use UCMs. This paper introduces patterns which provide guidance in selecting one of three major UCM styles depending on your software development context. The "Individual Maps" UCM style is most useful for rapidly and independently capturing a few key features of your system, features being optional or incremental units of functionality. The "Standard Root Map" UCM style is most appropriate if a small, evolving system consisting of interacting features needs to be documented. The "Isolation and Integration" UCM style is best applied to large, evolving systems with many interacting features.

1 Introduction

This document consists of two major parts. Section 2 "Background Information" gives a basic overview of a) the application domain of the patterns, i.e. capturing functional requirements and high level designs for complex systems composed of multiple features with a technique called Use Case Maps (UCMs) and b) the framework used to reason about forces, i.e. the NFR (Non-Functional Requirements) Framework. Readers familiar with Use Case Maps may skip the respective sub-section and proceed directly to section 2.2. Readers familiar with the NFR framework, however, are encouraged to read through section 2.2 since slightly different notational elements are used than suggested by the NFR Framework.

Sections 3 to 9 contain the actual collection of patterns organized in pattern form. Section 3 gives an overview of the pattern collection whereas sections 4 to 9 describe the patterns. These patterns provide guidance in selecting one of three UCM styles for various contexts.

2 Background Information

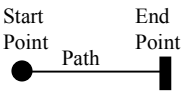
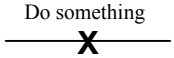
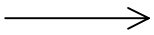
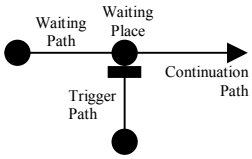
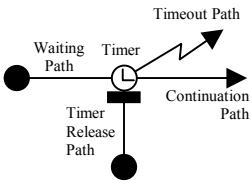
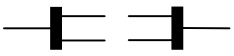
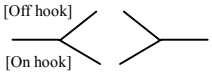
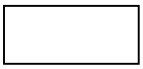
This section introduces the reader to the basic concepts and the notation of Use Case Maps – the application domain of the patterns presented in sections 3 to 9 – as well as the NFR framework used to reason about forces. The material in section 2 covers only those aspects of Use Case Maps and the NFR Framework required for the patterns. For further information on UCMs and the NFR framework see Section 11 "References" at the end of the document.

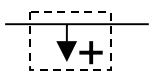
---

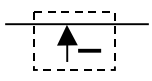
[1] The majority of this work was conducted while the authors were at Mitel Networks, 350 Legget Dr., Kanata (ON), Canada, K2K 2W7.

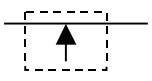
2.1 Use Case Maps Concepts and Notation

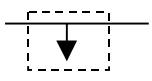
Use Case Maps (UCMs) [8, 9] are a scenario-based software engineering technique most useful at the early stages of software development. UCMs visually represent the causal relationships of responsibilities of one or more use cases combined with structures in one single view. The relationships are said to be causal because they involve concurrency and partial orderings of responsibilities, because they link causes to effects, and because they abstract from component interactions expressed by message exchanges. UCMs show related use cases in a map-like diagram. The map shows the progression of scenarios along use cases.

UCM Notation	Notation Explanation
	**Basic path.** The basic path is the most basic, complete unit. The path represents scenario flow. Paths connect start points, responsibilities, and end points. A path may have any shape as long as it is continuous (can cross itself). The start points represent preconditions or triggering causes. The end points represent post-conditions or resulting effects.
	**Responsibility point.** Represents generic processing (actions, tasks, or functions to be performed). Responsibilities may be bound to a component.
	**Direction** (optional). In general, the positioning of the start and end points of a path indicate direction. In certain cases, it is useful to show the direction on a complicated map.
	**Waiting place.** Represents a waiting place along a path. Propagation along the path stops at the waiting place until the trigger arrives. Waiting places can be triggered by a trigger path as shown or by the environment.
	**Timer.** A special waiting place that expresses the idea that there is a time limit on waiting. When propagation along the waiting path reaches the timer, the timer is set. Propagation along the continuation path continues if the timer release arrives. Propagation along the timeout path continues if the timeout occurs.
	**AND Fork** and **AND Join.** For concurrent paths (two or more).
	**OR Fork** and **OR Join.** An OR fork indicates that the path proceeds in only one out of two or more directions. Labels may identify alternative paths or guarding conditions. An OR Join indicates a common causal segment of two or more paths.
	**Static stub.** Contains only one plug-in (sub UCM), hence enabling hierarchical decomposition of complex maps.
	**Dynamic stub.** May contain several plug-ins, whose selection can be determined at run-time according to a *selection policy* (often described with pre-conditions).
	**Generic component.** Represents an architectural entity.
	**Slot.** Placeholder for dynamic components as operational units. **Dynamic responsibilities** can move dynamic components from a path into a slot or out of a slot onto a path.







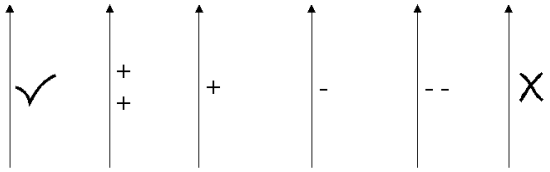


UCMs have a history of applications to the description of reactive systems of different natures (e.g. [2, 3, 4, 10]), to the avoidance and detection of undesirable interactions between scenarios or services (e.g. [4, 10, 18]), to early performance analysis (e.g. [20]), and the generation of more detailed behavioral models (e.g. [17, 19]). The UCM notation is also being considered by ITU-T for describing functional requirements as part of the upcoming User Requirements Notation standard [11]. The table above covers only the notational aspects of UCMs as required for sections 3 to 9. For further information on UCM concepts and the notation, the reader is referred to the virtual library of the *UCM User Group* [21].

The *UCM Navigator* tool [16, 17] supports a superset of the notational elements listed above as well as navigation through UCMs. The tool was used to create the UCMs in this document.

2.2 NFR Framework for Reasoning about Forces

The technique used to illustrate the forces of the context and the impact of solutions on these forces is based on the *NFR framework* [12, 15] and on work on the combination of the NFR framework and patterns [22]. The *OME* tool [14], developed at the Knowledge Management Lab at the University of Toronto, allows the creation of force graphs. The following table covers only the notational aspects of the framework as instantiated for the pattern domain.

<table>
<tr><td>Force</td><td> 
</td><td>Solution</td><td> 
</td></tr>
<tr><td>Contribution Links</td><td colspan="3"> 

Contribution links connect solutions or forces (source) and forces (target). The links indicate the impact a source has on a target.
✓ The source impacts the target so that the target is sufficiently balanced.
++ The source balances the target but not sufficiently.
+ The source balances the target but less than ++.
– The source unbalances the target but less than – –.
– – The source unbalances the target but not completely.
✗ The source impacts the target so that the target is completely unbalanced.
(Note that this notation uses different labels than those in the NFR framework.)</td></tr>
<tr><td>Force/Solution Labels</td><td colspan="3">A label positioned next to a force indicates how balanced the force is.
A label positioned next to a solution indicates how much of the solution has been achieved.
✓ Balanced/Fully achieved
+ Weakly balanced/achieved
0 Less than weakly balanced/achieved but more than weakly unbalanced/not achieved
– Weakly unbalanced/not achieved
✗ Unbalanced/Not achieved
(Note that this notation uses different labels than those in the NFR framework.)</td></tr>
</table>

3 A Collection of Patterns for Use Case Maps

This section gives a general introduction to the collection of patterns including general discussions on the overall context and problem for the pattern collection and the forces relevant to the pattern collection.

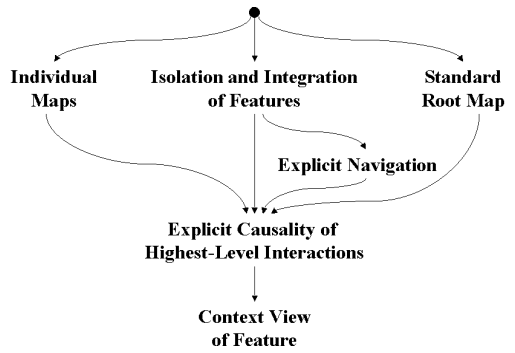


Figure 1: A Collection of Patterns for UCMs

3.1 Introduction

The collection of patterns consists of six patterns as depicted in Figure 1. The confidence in these patterns is very high because the patterns have been observed for numerous features in a large and complex call control software system for a telephone switch. Different development teams have developed these features over years. This paper fully describes the three top-level patterns and gives patlets for the remaining three patterns. Four additional patterns have been identified which extend this collection. These patterns, however, will not be included in this document in consideration of review time and the length of the document.

3.1.1 Overall Context

The overall context of the pattern collection at the root of Figure 1 is defined as follows. You are capturing functional requirements and high-level designs of your system. A scenario-driven software development approach has been chosen for your system. Furthermore, it has been decided that Use Case Maps (UCMs) will be used to capture functional requirements and high level designs. The UCM Navigator provides tool support. Thus, any solution has to consider the constraints of the tool.

UCMs give software engineers a high degree of freedom in how to describe systems. Currently, three UCM styles are known. Literature often suggests the “Individual Maps” [3, 5] approach (see 4) and the “Standard Root Map” [6, 13, 17, 18] approach (see 5). This paper introduces the “Isolation and Integration” approach (see 6). It, however, is not clear whether these approaches are applicable to all sub-contexts of the overall context and, if not, to which sub-contexts they are applicable.

3.1.2 General Problem

Find the best UCM style to be used in the given context.

The problem addressed by the discussed patterns does not relate directly to the use of UCMs for the description of behavioral patterns or design patterns. It relates to how general scenario descriptions (use cases, features) can be best organized in various contexts with the help of the UCM notation. The problem is also not related to whether UCMs are an appropriate technique for capturing requirements and high level designs. These issues have been addressed elsewhere [1, 7, 8, 9].

3.1.3 General Discussion on Forces

Figure 2 shows the hierarchy of forces that have to be considered for the overall context.

a) Evolveability:

An important high-level concern is the evolveability of the system. Several sub-forces contribute to the evolveability of a large system:

a.1) Understandability of a single feature:

A clear understanding of new and existing features is the basis for evolveability. The understandability of a feature depends on:

a.1.1) the simplicity of each UCM required for the feature (the simpler, the better),

a.1.2) the number of hierarchical levels of UCMs per feature (the lower, the better),

a.1.3) the number of distributed UCMs, i.e. the UCMs that belong to the same feature but are disjoint (the lower, the better),

a.1.4) the pollution of the feature description, i.e. whether UCM paths and path elements from different features can be clearly identified (the less polluted, the better), and

a.1.5) the ability to detect undesirable feature interactions [23] and specify desirable feature interactions (the easier, the better). In general, feature interaction is a key force for the understandability of features.

a.1.5.1) Connected and consistent feature descriptions are the principal prerequisite for feature interaction detection and specification.

a.2) Scaleability:

Another important aspect for the evolveability of the system is the scaleability of the chosen UCM style to a large system with hundreds of features. The scaleability depends foremost on connected and consistent feature descriptions (see a.1.5.1)) and to a lesser extent on the distribution and pollution of feature maps (see a.1.3) and a.1.4), respectively).

a.3) Reuseability:

To a lesser extent, the reuseability of UCMs across a number of features contributes positively to the evolveability of the system.

a.4) Ability to specify test cases:

To a lesser extent, the ability to use UCMs for the specification of test cases for single features and feature combinations contributes positively to the evolveability of the system. The ability to specify test cases for feature combinations depends on connected and consistent feature descriptions (see a.1.5.1)).

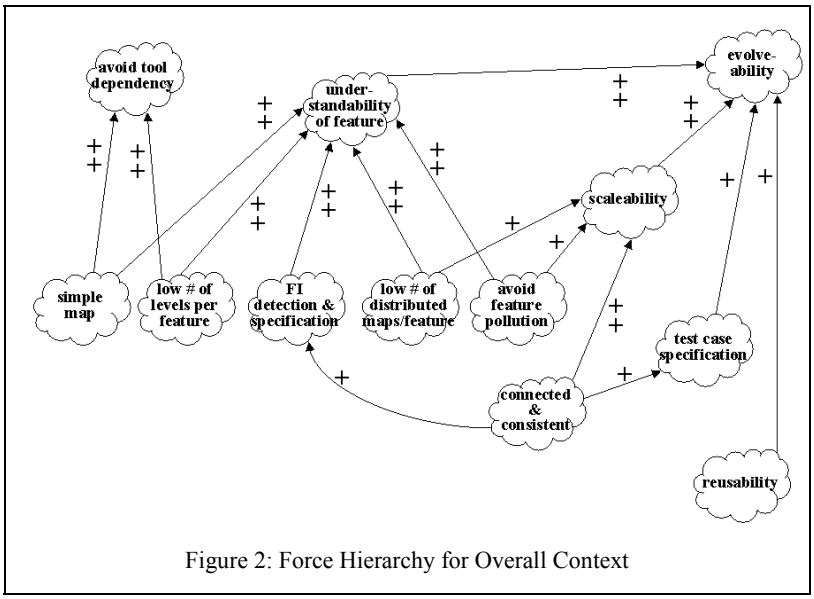


Figure 2: Force Hierarchy for Overall Context

b) Tool dependency:

A second issue to keep in mind is the tool dependency of the chosen UCM

style with respect to the distribution of UCMs to non-authors such as reviewers and new team members. UCMs can be read easily without the tool if it is easy to navigate through a series of UCMs. The ease of navigation relies on the simplicity of UCMs (in terms of the UCM path itself and the number of stubs per feature) (see a.1.1)) and the number of levels of UCMs per feature (see a.1.2)).

In general, most forces are connected with each other in more or less subtle ways. The force hierarchy shows only the important relationships which impact decisions. Some subtle connections, however, exist in addition to the ones shown and are addressed, if necessary, in the discussion of forces for each pattern.

3.1.4 Summary of Known Uses and Examples

All examples used in the following sections are taken from a call control software system but have been simplified and modified to protect confidential material. The changes have been carefully made as not to distort the occurrences of the discussed patterns. Although this paper focuses on examples from the telecommunication domain and cites known uses mainly from the same domain, we know of no reason why the patterns would not apply to other complex domains as well.

4 Individual Maps

Name of Pattern	Individual Maps
Brief Description	Independent feature descriptions are the best UCM style to be used in a context where the understandability of single features and tool independence are very important and the ability to specify test cases also has to be considered.
Confidence in Pattern	*** (out of *, **, or ***)
Discovered By	Ray Buhr
Authors	Gunter Mussbacher, Daniel Amyot
Shepherd	Jim Coplien
Other Reviewers	Rossana Andrade, Tom Gray, Michael Weiss
Date	October 5, 2001

4.1 Context

You are capturing functional requirements and high-level designs of a *few* key features of your system in order to get an initial understanding of the system and early feedback from stakeholders. Since you are creating a prototype rather than a complete specification, you will *not* be concerned if features interact with each other (i.e. one feature may impact the behavior of another feature either in a desirable way or in some unexpected or undesirable way). There is *no* need to reuse parts of features or even evolve a whole system efficiently but it must be possible to change single features *rapidly* and *independently* of other features. Often a pen and paper or a whiteboard rather than a tool are used to quickly create sketches of your prototype.

4.2 Problem

What is the best UCM style to be used in the given context?

4.3 Forces

All forces of the pattern collection have been described in 3.1.3. In the context of the "Individual Maps" pattern only some of these forces are relevant as indicated in Figure 4 to Figure 6. Note that the feature distribution, pollution, interaction, reusability, and scalability forces are not considered in this context because these five forces are not an issue if only a few features are captured.

If the "Standard Root Map" UCM style is applied in this context, the forces will not be sufficiently balanced (see Figure 4) even though the understandability force is reasonably well balanced.

- The understandability of a single feature is not as good as it could be because a higher number of stubs causes the maps to be a little bit more complex and at least one more level (the root level) is introduced as an additional level for all features.
- More importantly, tool dependency has increased because the stub-rich root map requires significantly more jumps between maps.
- It is possible but cumbersome to specify test cases for single features as definitions of features are buried in the map hierarchy.

If the "Isolation and Integration" UCM style is applied in this context, the forces are even less sufficiently balanced than with the "Standard Root Map" UCM style (see Figure 5). Both, understandability of single features and tool independence, are affected by the higher complexity required for the "Isolation and Integration" UCM style.

4.4 Solution

The "Individual Maps" UCM style describes each feature individually (see Figure 3). A hierarchy of one or more UCMs may be used to describe a feature but each UCM belongs solely to the described feature and is not reused for any other feature.

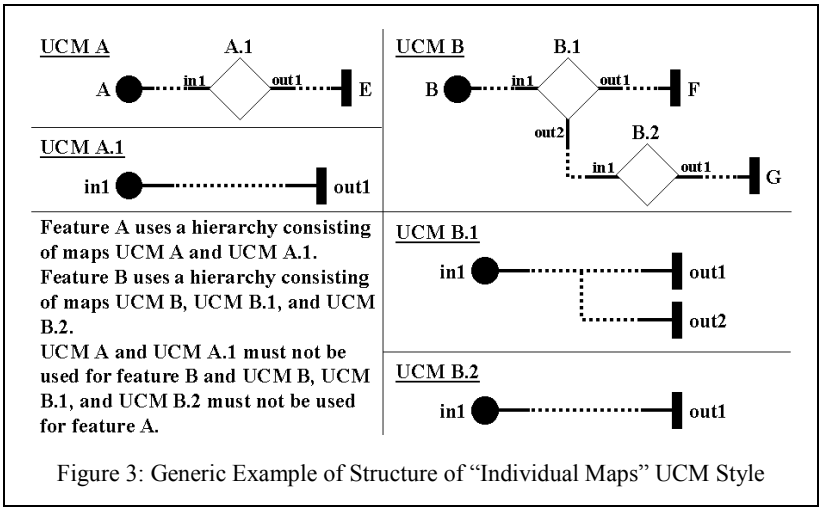


Figure 3: Generic Example of Structure of "Individual Maps" UCM Style

4.5 Resulting Context

Figure 6 shows how the solution balances the forces identified in 3.1.3. The "Individual Maps" UCM style balances well all relevant forces in the context because:

- The simplicity of UCMs and the number of levels of UCMs per feature are well balanced. This is as good as any approach can get since UCMs always have to trade-off simple

UCMs with additional levels of UCMs (i.e. number of stubs and plug-ins). Thus, the understandability of single features is perfectly balanced.

- Tool dependency is kept low because reading UCMs without the tool requires only the minimum amount of jumps between different levels of UCMs due to the optimal number of UCM levels.
- Test cases for single features can be specified quite easily due to the clear, unpolluted, and complete description of each feature.

Although the "Individual Maps" UCM style balances well tool dependency and understandability forces, this approach does not show the causal relationship between user actions (e.g. Figure 7 does not define whether *directory number* or *end call* can occur before *start call*). The pattern "Explicit Causality of Highest-Level Interaction" in section 8 addresses this problem.

As a consequence of using the "Individual Maps" UCM style, one cannot expect other concerns relevant in the overall context of the pattern language but not relevant in the sub-context of this pattern to be addressed.

- The style cannot detect

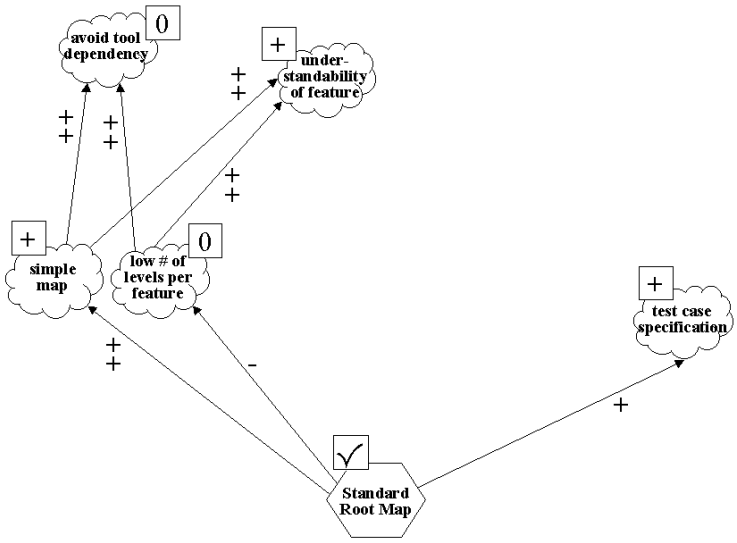


Figure 4: Impact on Individual Maps Forces by Standard Root Map

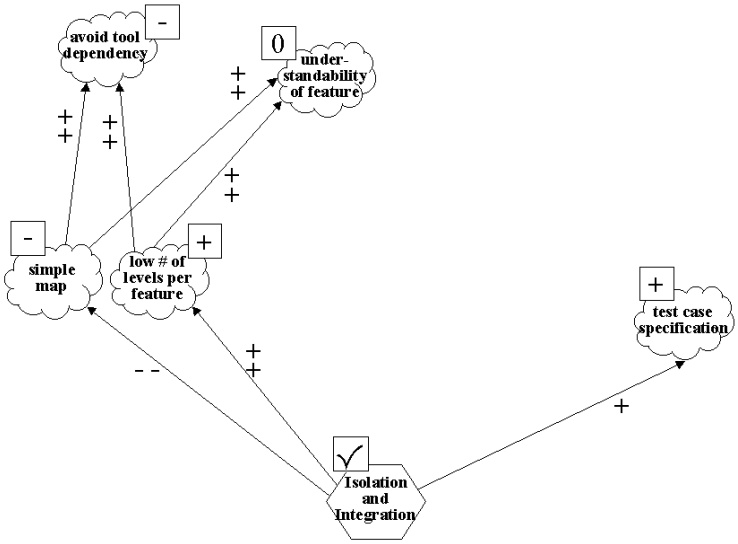


Figure 5: Impact on Individual Maps Forces by Isolation & Integration

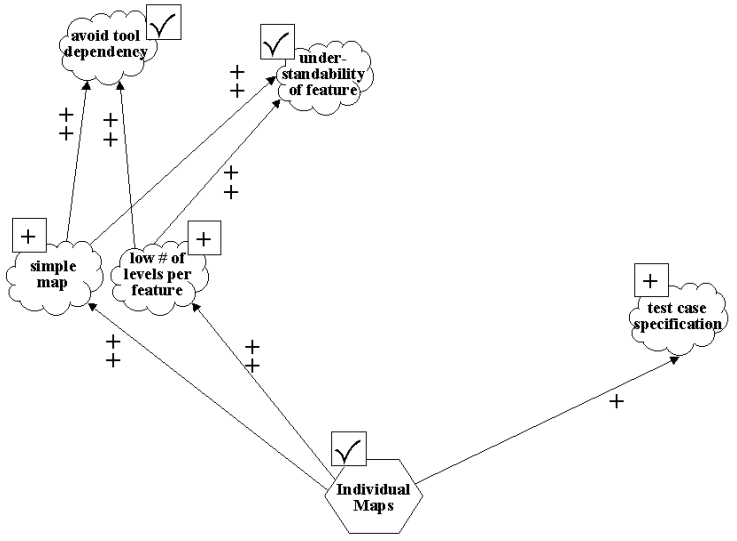


Figure 6: Impact on Individual Maps Forces by Individual Maps

interactions among features because consistency across features is not enforced. The style also cannot specify feature interactions because the definitions of different features are not linked.

- Test cases for feature combinations cannot be specified for the same reasons.
- Per definition of the style, reuse cannot be achieved.
- The approach does not scale well. Feature definitions are not linked thus more and more inconsistencies will potentially be introduced with each new feature. Although each new feature by itself requires only a similar effort to document than previous features, the amount of time eventually spent to work out the inconsistencies greatly increases.

4.6 Known Uses

For further examples of the "Individual Maps" UCM style in addition to the ones shown in Section 4.7 see [3, 5].

4.7 Examples of "Individual Maps" UCM Style

Examples of two features documented using the "Individual Maps" UCM style are shown in Figure 7 (Basic Call) and Figure 8 to Figure 10 (Automatic Call Distribution). Both examples use an underlying call model based on originating and terminating call halves.

4.7.1 Basic Call

The Basic Call feature (Figure 7) starts at the *start call* start point. The path then waits at the *dialing* waiting place until a directory number is entered (*directory number* start point). If a wrong number is entered the path takes a left turn and ends at *fail*. If a correct number is entered, the terminating call half is created (*create_TCH*).

The rest of the scenario depends on the device's availability. If the device is busy, the terminating call half follows the path labeled *[busy]*, ends at *idle.t*, and in parallel informs the originating call half which applies *busy* tone and ends at *fail*. If the device is not busy, the device rings, the terminating call half follows the path labeled *[not_busy]*, then waits at the *wfa.t* waiting place, and in parallel informs the originating call half which applies *ringback* tone and sets the *wait for answer* timer.

What comes next depends on whether the callee answers (*answer* start point) before the *wait for answer* timer expires. If the timer expires, the originating call half ends at *fail* and in parallel informs the terminating call half which follows the *[ring_timeout]*

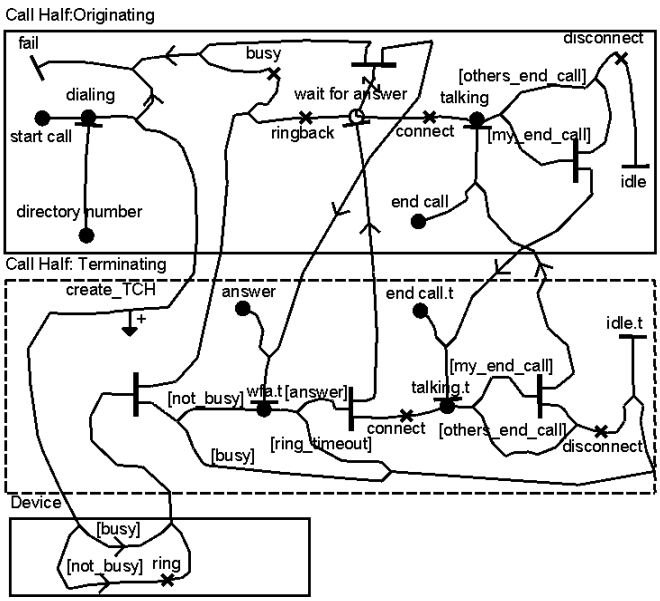


Figure 7: Basic Call (Individual Maps)

path and ends at *idle.t*. If the callee answers, the terminating call half follows the *[answer]* path, *connects*, and then waits at the *talking.t* waiting place. In parallel, the terminating call half informs the originating call half which *connects* and waits at the *talking* waiting place.

The rest of the scenario depends on whether the caller or the callee ends the call. If the caller ends the call (*end call* start point), the originating call half follows the *[my_end_call]* path, *disconnects*, ends at *idle*, and in parallel informs the terminating call half which follows the *[others_end_call]* path, *disconnects*, and ends at *idle.t*. If the callee ends the call (*end call.t* start point), the terminating call half follows the *[my_end_call]* path, *disconnects*, ends at *idle.t*, and in parallel informs the originating call half which follows the *[others_end_call]* path, *disconnects*, and ends at *idle*.

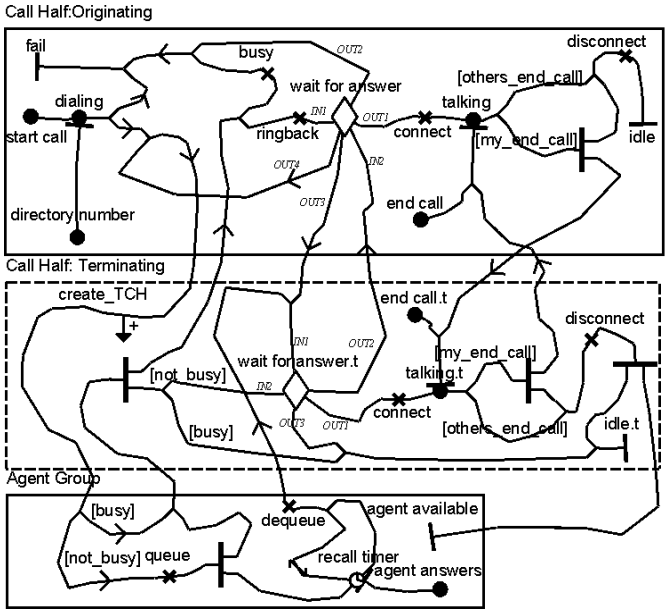


Figure 8: ACD (Individual Maps)

4.7.2 Automatic Call Distribution (ACD)

ACD distributes incoming calls to a group of agents according to their availabilities. The ACD feature (Figure 8, Figure 9, and Figure 10) duplicates a lot of the Basic Call feature. Therefore, only the changes to Basic Call behavior will be described. The biggest difference is that an agent group has replaced the device. In the case of a busy agent group, the agent group just informs the terminating call half. If the agent group is not busy, the agent group additionally *queues* the call and sets the *recall timer*.

The behavior of the originating and terminating call half is exactly the same as Basic Call behavior until the originating and terminating call halves are waiting at the *wait for answer* and *wfa.t* waiting places, respectively. The waiting places have been moved to two plug-in maps due to space constraints and readability concerns. The in and out labels on the UCMs show the binding between the parent map (ACD) and the plug-in maps (Wait For Answer, Wait For Answer.t).

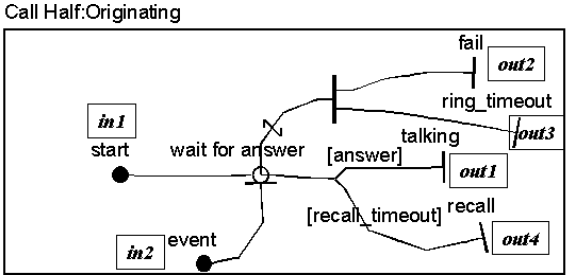


Figure 9: Wait For Answer (Individual Maps – ACD)

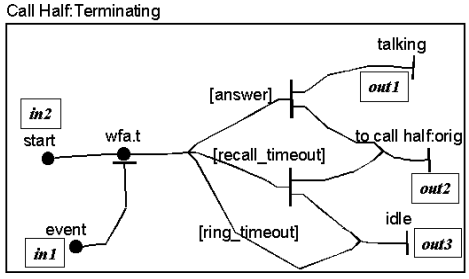


Figure 10: Wait For Answer.t (Individual Maps–ACD)

What comes next depends on whether the agent answers *(agent answers* start point) before the *recall* timer or the *wait for answer* timer expires. If the recall timer expires before the agent answers, the agent group *dequeues* the call and informs the terminating call half. This terminating call half follows the *[recall_timeout]* path (see Figure 10), ends at *idle.t*, and in parallel informs the originating call half which follows the *[recall_timeout]* path (see Figure 9) and tries to *recall* (i.e. the path loops back from the *wait for answer* stub and a new terminating call half is created (*create_TCH*)). If the agent answers, the agent group *dequeues* the call and informs the terminating call half. The terminating and originating call halves then follow normal Basic Call behavior.

Please note that the *wait for answer* timer is always set to a longer duration than the *recall* timer is. In case the wait for answer timer expires before the agent answers, a fatal error of the agent group may be assumed and normal Basic Call behavior occurs.

The last variation of Basic Call behavior occurs when either the caller or the agent *ends the call*. In addition to Basic Call behavior, the terminating call half informs the agent group after *disconnecting* that the *agent* is now *available* again.

5 Standard Root Map

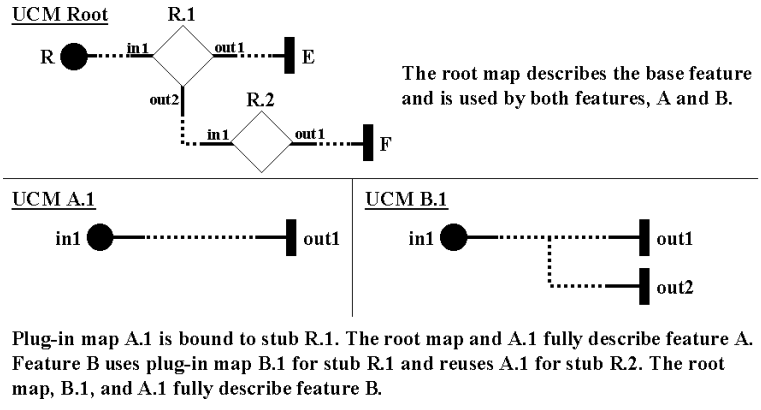
Name of Pattern	Standard Root Map
Brief Description	A combination of a root map, stubs, and plug-in maps is the best UCM style to be used in a context where evolveability, understandability of single features, and the ability to detect, specify, and understand feature interactions are very important, and reuseability across various features, the ability to specify test cases, and tool dependency also have to be considered.
Confidence in Pattern	*** (out of *, **, or ***)
Discovered By	Ray Buhr
Authors	Gunter Mussbacher, Daniel Amyot
Shepherd	Jim Coplien
Other Reviewers	Rossana Andrade, Tom Gray, Michael Weiss
Date	October 5, 2001

5.1 Context

You are capturing functional requirements and high-level designs of a *small* system that will be *evolving* over a long time or has been evolving for a long time. The features of the system may be *interacting* thereby increasing the system's complexity. Feature interaction occurs if one feature impacts the behavior of another feature either in a desirable way or in some unexpected or undesirable way. A base feature defining the framework for all other features usually but not exclusively indicates feature interactions. Most of the time new features are variations of existing features and build on the base feature and other features.

5.2 Problem

What is the best UCM style to be used in the given context?



Plug-in map A.1 is bound to stub R.1. The root map and A.1 fully describe feature A. Feature B uses plug-in map B.1 for stub R.1 and reuses A.1 for stub R.2. The root map, B.1, and A.1 fully describe feature B.

Figure 11: Generic Example of Structure of "Standard Root Map" UCM Style

5.3 Forces

All forces of the pattern collection have been described in 3.1.3. In the context of the "Standard Root Map" pattern only some of these forces are relevant as indicated in Figure 12 to Figure 14. Note that the feature distribution, pollution, and scaleability forces are not considered in this context because these three forces are not an issue if only a small system is captured.

If the "Individual Maps" UCM style is applied in this context, the forces will not be sufficiently balanced (see Figure 12) even though the tool dependency force is perfectly balanced.

- The style cannot detect interactions among features because consistency across features is not enforced. The style also cannot specify feature interactions because the definitions of different features are not linked. Therefore, the understandability force is not balanced sufficiently although feature maps are simple and the number of levels is low.
- Per definition of the style, reuse cannot be achieved.
- With mediocre understandability of features, no reuse, and test cases only for single features, evolveability is not balanced at all.

If the "Isolation and Integration" UCM style is applied in this context, the forces will not be sufficiently balanced (see Figure 13) even though the evolveability force is reasonably well balanced.

- The style is more dependent on tool support because of the complexity of the maps and the frequent use of stubs.
- The increased complexity of feature maps makes it more difficult to understand features although feature interaction detection and specification is possible and the number of levels per feature is kept low.
- Although consistency is enforced by the integration of features, the reuseability force is not as well balanced as it could be since more than one reusable unit may exist on a single UCM.

To summarize, a perfect solution should balance the tool dependency force and sub-forces a.1.1) and a.1.2) of the understandability force as the "Individual" UCM style does. Furthermore, a perfect solution should balance the feature interaction and specification force as the "Isolation and Integration" UCM style does but further improve the reusability and map complexity forces.

5.4 Solution

The "Standard Root Map" UCM style describes the base feature on a UCM called the root

map which contains several stubs (see Figure 11). The root map and the default set of plug-in maps for the stubs define the base feature. Other features may use one or more different plug-in maps and thus vary the behavior of the base feature. Since stubs may be used at any level of the map hierarchy, new features may introduce variations also at any level.

5.5 Resulting Context

Figure 14 shows how the solution balances the forces identified in 3.1.3. All in all, the "Standard Root Map" UCM style balances well the evolveability force at the expense of a slight increase in the complexity of UCMs and thus tool independence. The evolveability force is sufficiently balanced because:

- Feature interaction detection and specification is possible since all features are connected to each other even though it is cumbersome because feature definitions are buried in the map hierarchy.
- The understandability of a single feature is sufficiently balanced because feature interaction detection and specification is possible and feature maps are only slightly more complex.

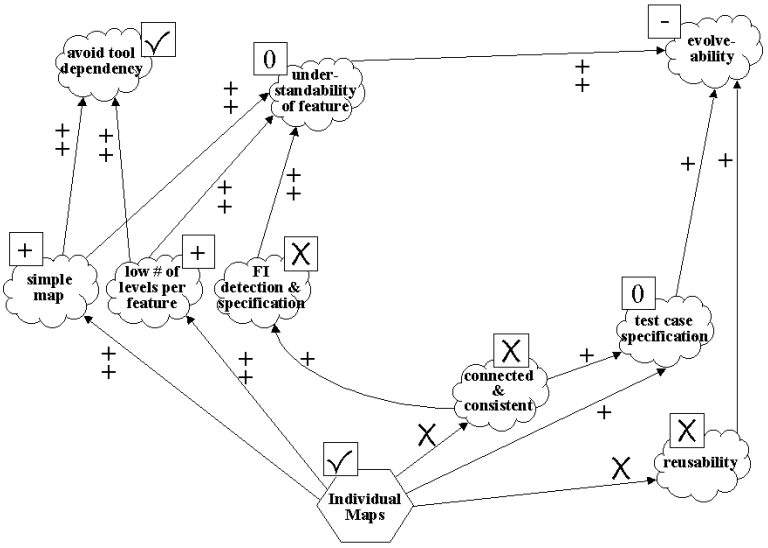


Figure 12: Impact on Standard Root Map Forces by Individual Maps

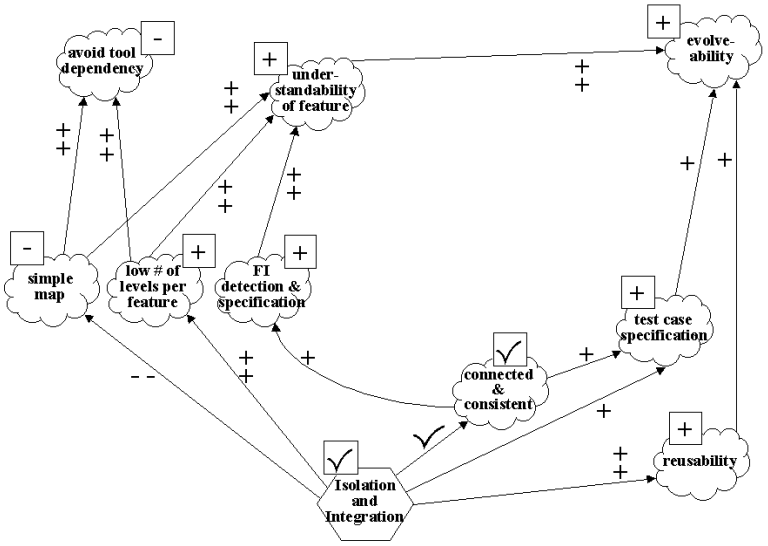


Figure 13: Impact on Standard Root Map Forces by Isolation & Integration

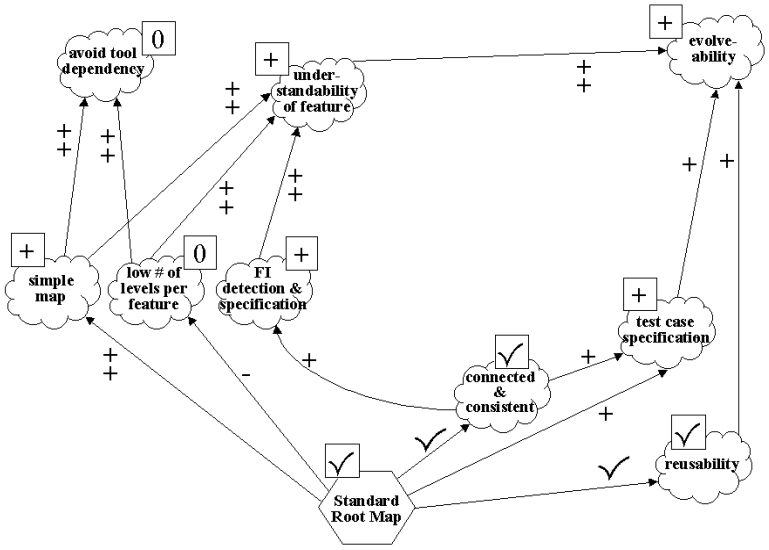


Figure 14: Impact on Standard Root Map Forces by Standard Root Map

- Reuseability can be achieved through the use of stubs with each plug-in map being one potential reusable unit.

Although the "Standard Root Map" UCM style balances well the evolveability force, this approach is more dependent on tool support than the "Individual Maps" UCM style because the stub-rich root map requires significantly more jumps between maps. The simplicity of UCMs is a little worse compared to the "Individual Maps" UCM style because the number of stubs has increased and the number of levels per feature has also increased by at least one level (the root level). Tool support and understandability, however, are still balanced better than with the "Isolation and Integration" UCM style.

This approach also does not show the causal relationship between user actions (e.g. Figure 15 does not define whether *directory number* or *end call* can occur before *start call*). The pattern "Explicit Causality of Highest-Level Interaction" in section 8 addresses this problem.

As a consequence of using the "Standard Root Map" UCM style, one cannot expect other concerns relevant in the overall context of the pattern language but not relevant in the sub-context of this pattern to be addressed.

- Feature maps do get polluted and are distributed which will cause problems if the system increases in size. Therefore, scaleability is not optimal yet although it has improved compared to the "Individual Maps" UCM style due to a greater degree of consistency.

5.6 Known Uses

For further examples of the "Standard Root Map " UCM style in addition to the ones shown in Section 5.7 see [6, 13, 17, 18].

5.7 Examples of "Standard Root Map" UCM Style

Examples of two features documented using the "Standard Root Map" UCM style are shown

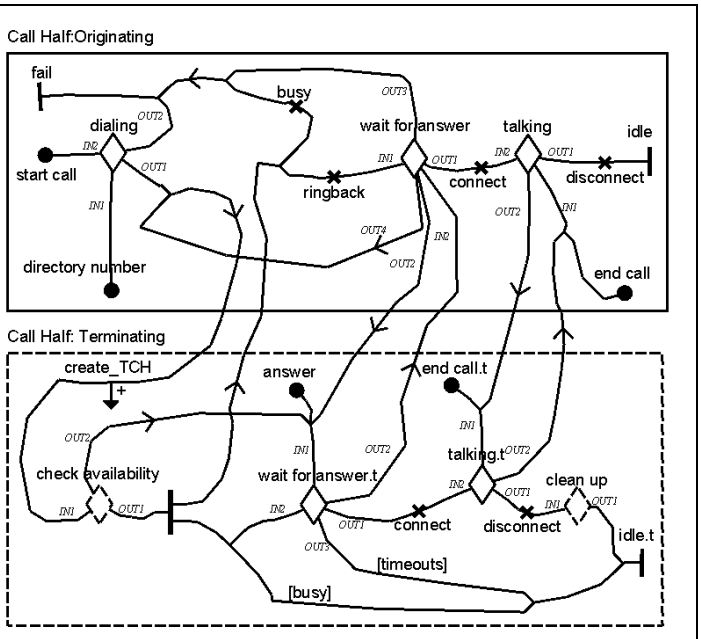


Figure 15: Basic Call root map (Standard Root Map)

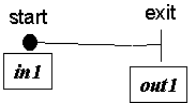


Figure 16: Nothing
(Standard Root Map – Basic Call)

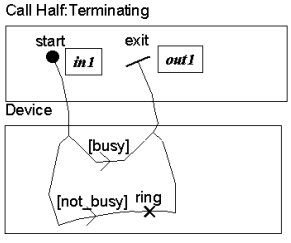


Figure 17: Check Availability
(Standard Root Map – Basic Call)

in Figure 15 to Figure 21 (Basic Call) and Figure 22 to Figure 25 (ACD). Both features start from the same root map (Figure 15).

5.7.1 Basic Call

The root map (Basic Call – Figure 15) and its six default plug-in maps (Dialing, Check Availability, Wait For Answer, Wait For Answer.t, Talking, Nothing) specify exactly the same behavior as explained for the "Individual Maps" UCM style (see 4.7.1).

Once again, in and out labels specify the bindings between stubs on the Basic Call map and their plug-in maps. The following list defines which stubs in Figure 15 contain which plug-in maps:

- the *check availability* stub contains the *Check Availability* plug-in (see Figure 17),
- the *dialing* stub contains the *Dialing* plug-in (see Figure 18),
- the *talking* and *talking.t* stubs contain the *Talking* plug-in (see Figure 19), and
- the *wait for answer* stub contains the *Wait For Answer* plug-in (see Figure 20),
- the *wait for answer.t* stub contains the *Wait For Answer.t* plug-in (see Figure 21),
- finally the *wait for answer.2* stub in Figure 20, *wait for answer.t.2* stub in Figure 21, and *clean up* stub in Figure 15 all contain the *Nothing* plug-in (see Figure 16).

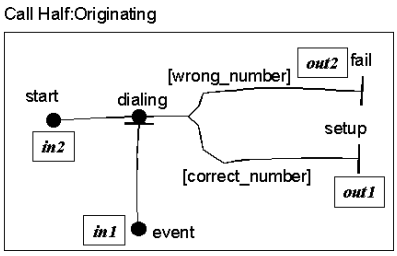


Figure 18: Dialing (Standard Root Map – Basic Call)

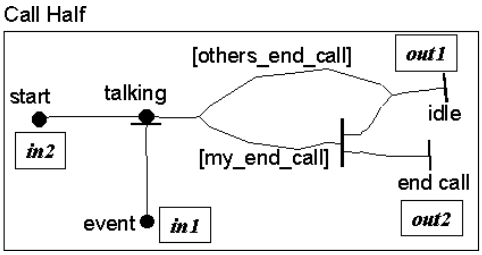


Figure 19: Talking (Standard Root Map – Basic Call)

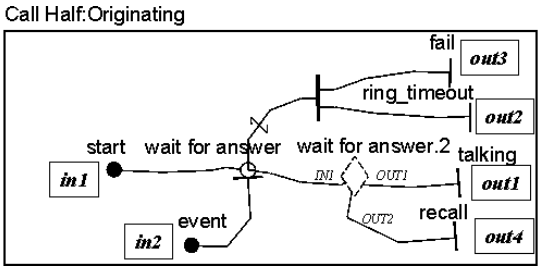


Figure 20: Wait For Answer
(Standard Root Map – Basic Call)

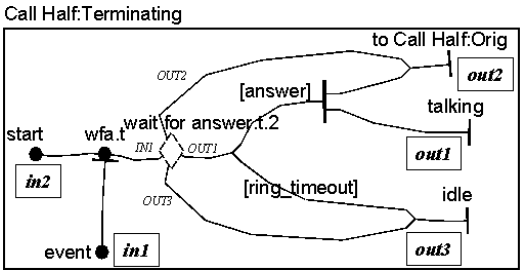


Figure 21: Wait For Answer.t
(Standard Root Map – Basic Call)

5.7.2 Automatic Call Distribution (ACD)

The new ACD plug-ins in conjunction with the Basic Call root map and plug-ins specify the exact same behavior as explained in the "Individual Maps" UCM style (see 4.7.2). The ACD feature overrides the Basic Call behavior by using the following four new plug-in maps:

- the *clean up* stub in Figure 15 contains the *Clean Up.ACD* plug-in (see Figure 22),
- the *check availability* stub in Figure 15 contains the *Check Availability.ACD* plug-in (see Figure 23),
- the *wait for answer.2* stub in Figure 20 contains the *Wait For Answer.ACD* plug-in (see Figure 25), and

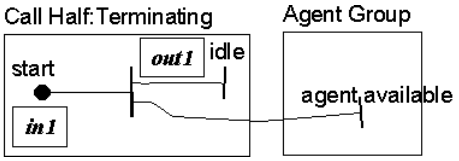


Figure 22: Clean Up.ACD
(Standard Root Map – ACD)

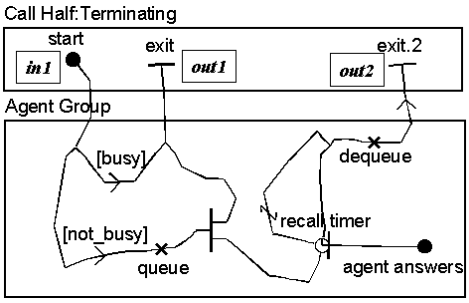


Figure 23: Check Availability.ACD
(Standard Root Map – ACD)

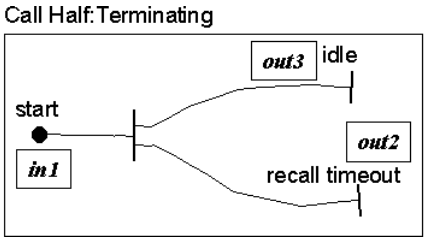


Figure 24: Wait For Answer.t.ACD
(Standard Root Map – ACD)

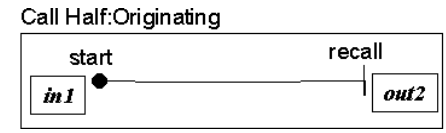


Figure 25: Wait For Answer.ACD
(Standard Root Map – ACD)

- the *wait for answer.t.2* stub in Figure 21 contains the *Wait For Answer.t.ACD* plug-in (see Figure 24).

6 Isolation and Integration of Features

Name of Pattern	Isolation and Integration of Features
Brief Description	Isolating features from each other but also linking them in a well-defined way is the best UCM style to be used in a context where evolveability, scaleability, understandability of single features, and the ability to detect, specify, and understand feature interactions are very important, and reuseability across various features, the ability to specify test cases, and tool dependency also have to be considered.
Confidence in Pattern	*** (out of *, **, or ***)
Discovered By	Gunter Mussbacher
Authors	Gunter Mussbacher, Daniel Amyot
Shepherd	Jim Coplien
Other Reviewers	Rossana Andrade, Tom Gray, Michael Weiss
Date	October 5, 2001

6.1 Context

You are capturing functional requirements and high-level designs of a *large* system that will be *evolving* over a long time or has been evolving for a long time. The system consists of *many interacting* features increasing its complexity. Feature interaction occurs if one feature impacts the behavior of another feature either in a desirable way or in some unexpected or

undesirable way. A base feature defining the framework for all other features usually but not exclusively indicates feature interactions. Most of the time new features are variations of existing features and build on the base feature and other features.

6.2 Problem

What is the best UCM style to be used in the given context?

6.3 Forces

All forces of the pattern collection have been described in 3.1.3. In the context of the "Isolation and Integration of Features" pattern all of these forces are relevant as indicated in Figure 26 to Figure 28.

If the "Individual Maps" UCM style is applied in this context, the forces will not be sufficiently balanced (see Figure 26) even though the tool dependency force and sub-forces a.1.1), a.1.2), a.1.3), and a.1.4) of the understandability force are well balanced.

- The style cannot support the detection of interactions among features because consistency across features is not enforced. The style also cannot specify feature interactions because the definitions of different features

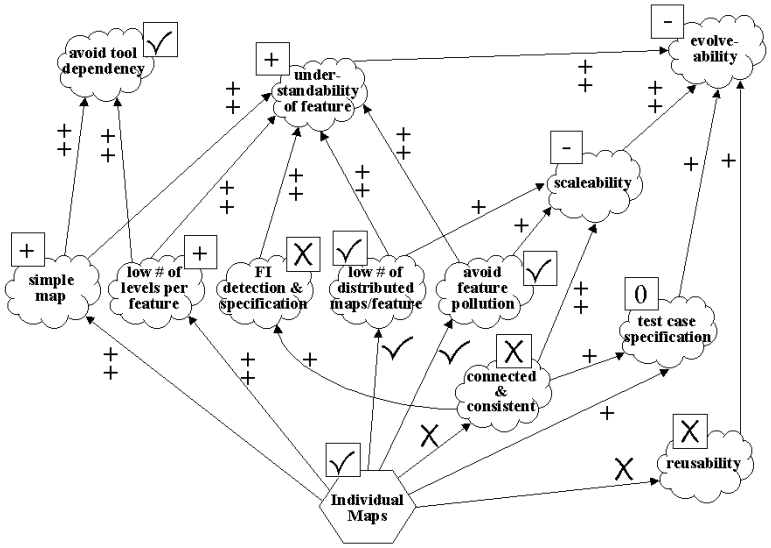


Figure 26: Impact on Isolation & Integration Forces by Individual Maps

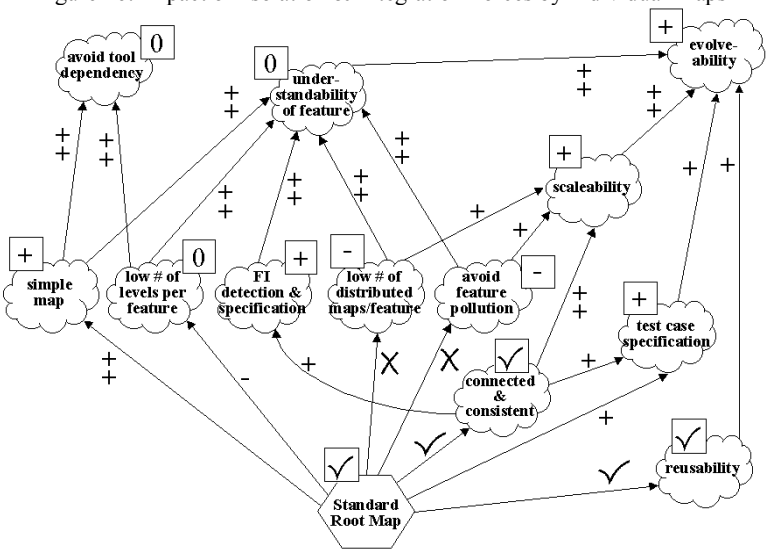


Figure 27: Impact on Isolation & Integration Forces by Standard Root Map

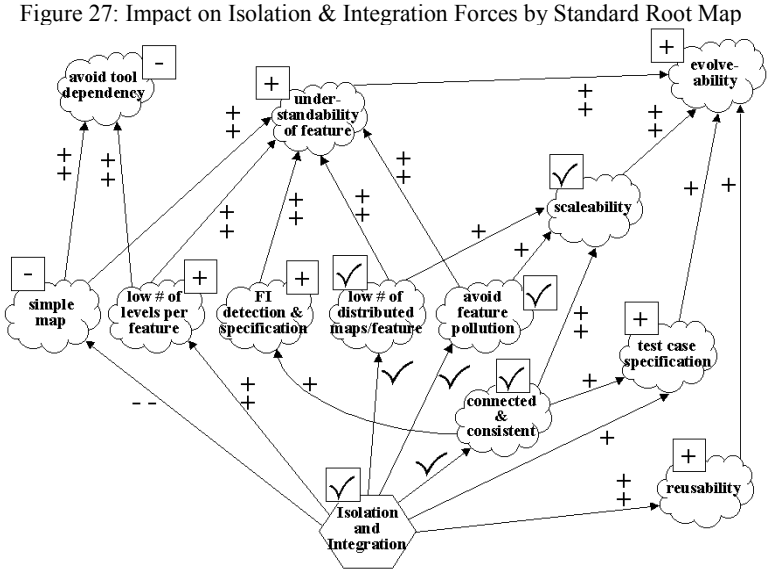


Figure 28: Impact on Isolation & Integration Forces by Isolation &Integration

are not linked. Therefore, the understandability force is only weakly balanced.
- Per definition of the style, reuse cannot be achieved.
- Test cases for feature interactions cannot be specified.
- The approach also does not scale well. Feature definitions are not linked thus more and more inconsistencies will potentially be introduced with each new feature. Although each new feature by itself requires only a similar effort to document than previous features, the amount of time eventually spent to work out inconsistencies greatly increases.

If the "Standard Root Map" UCM style is applied in this context, the forces will not be sufficiently balanced (see Figure 27) even though the reuseability force is perfectly balanced and the balance of the scaleability and feature interaction and detection forces is improved.

- Feature maps do get polluted now since all new features are plug-ins of at least the base root map. Therefore, variations of the base behavior caused by these new features do show up in the root map which should only show the base feature. E.g. the loop back from the *wait for answer* stub in Figure 15, out-path 2 of the *check availability* stub in Figure 15, and some out-paths of the *wait for answer.2* and *wait for answer.t.2* stubs in Figure 20 and Figure 21, respectively, are not required for Basic Call but exist because of ACD. Considering the large number of features in the system, feature maps get so polluted that the original description of the feature is effectively lost.
- Feature maps are now distributed. E.g. the set of UCMs specific to the ACD feature includes the UCMs in Figure 22, Figure 23, Figure 24, and Figure 25. These four UCMs are completely disjoint (one cannot move directly from any UCM to another). In contrast, the three ACD feature maps from the "Individual Maps" style (Figure 8, Figure 9, and Figure 10) are directly connected to each other. The reason for the distributed feature maps is that everything has to go through the root map which belongs only to the base feature. Considering a large number of features, it becomes difficult to find all UCMs that belong to a given feature. A naming scheme could help but breaks down when UCMs are being reused for various features. The UCM Navigator's functionality to group maps into sets could be used but navigation through the set still remains difficult. The main starting point for a feature is also not as apparent as it is in the "Individual Maps" UCM style.
- Scaleability improves because consistency is enforced to a greater degree. Scaleability, however, is not optimal yet because pollution and distribution of features remain a problem in large systems.
- Feature interaction detection and specification is now possible since all features are connected to each other but is cumbersome since complete feature definitions are buried at various levels in the map hierarchy and the specification of feature interactions further pollutes the description of features.
- The understandability of a single feature suffers considerably since feature maps get polluted, feature maps are distributed, and at least one more level (the root level) has been introduced as an additional level for all features.
- Tool dependency has also increased because the stub-rich root map requires significantly more jumps between maps.

Overall, this UCM style contributes positively but not sufficiently to evolveability because of a trade-off of reuseability and moderate amounts of feature interaction detection and specification, scaleability, and test case specification against the overall understandability of a single feature.

To summarize, a perfect solution should balance the tool dependency force and sub-forces a.1.1), a.1.2), a.1.3), and a.1.4) of the understandability force as the "Individual" UCM style does. Furthermore, a perfect solution should balance the reuseability force as the "Standard Root Map" UCM style does but further improve the scaleability and feature interaction detection and specification forces.

6.4 Solution

The "Isolation and Integration" UCM style isolates features from each other but does not keep these features completely separate. The features are linked to each other in a well-defined way. First, a root map for each feature provides initial isolation. Second, specific UCM structures identify locations where a link between two features may exist, thus further isolating features. The *event stub* structure is used for locations where the system is ready to deal with an event that may not be related to the feature. The *feature interaction (FI) fork* structure is used for locations where a variation of the feature may occur and this variation is caused by another feature. Once these two kinds of locations have been isolated, it is possible to integrate features by referencing the isolated locations.

The remainder of this section explains in more detail the event stub structure, the FI fork structure, and the referencing mechanism.

6.4.1 Event Stub Structure

The event stub structure is inserted at the first kind of location (see Figure 29 and Figure 30). Note that the FI fork structure is part of the event stub structure. The binding between the stub and the plug-in is defined as follows: (IN1, start), (IN2, event), (OUT1, success), (OUT2, fail), and (OUT3, feature).

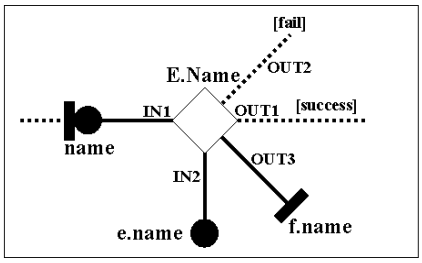


Figure 29: Structure of Event Stub

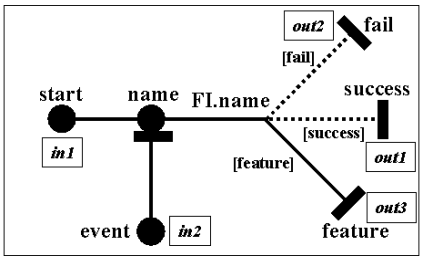


Figure 30: Structure of Plug-in for Event Stub

The event stub has two uses. First, if a UCM author wants to express that the current scenario is at the location represented by the event stub, path IN1 is taken. This positions the scenario at the *name* waiting place in the plug-in and the system is now waiting for an event. Second, if the UCM author wants to express that the scenario is continued because an event occurred, the source of the event is connected to the *e.name* start point and path IN2 is taken. This allows the scenario to continue to the FI fork labeled *FI.name*. At this point, the event that occurred is examined. If the event is known to the feature the event stub belongs to, one of the success or fail exits will be taken (one or more of these may be specified as required by the feature). If the event is unknown to the feature (i.e. another feature is changing the behavior of this feature – feature interaction!), then the feature exit will be taken. The FI fork is therefore defined as: "IF ((NOT success) AND (NOT

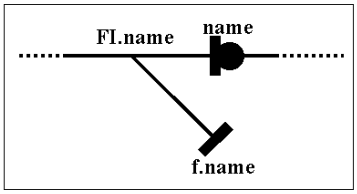


Figure 31: Structure of FI fork

fail)) THEN feature". Note that this definition is only dependent on the feature the event stub belongs to.

6.4.2 FI Fork Structure

At the second kind of location, only a subset of the event stub structure is required. Just the FI fork is added possibly followed by an end point connected to a labeled start point (see Figure 31). In this case, the definition of the FI fork explicitly states the reason for taking the feature exit (*f.name*) (e.g. "IF (ACD) THEN feature"). The FI fork indicates that the behavior of the feature is altered by another feature which will cause the scenario to exit at the feature exit. A guard (e.g. [feature]) may be used to label the branch to the feature exit (*f.name*) but is often omitted due to space constraints.

6.4.3 Referencing Mechanism

Features are integrated by referencing the isolated locations. The reference mechanism works as follows.

Each end point representing a feature exit implicitly sets a postcondition (see *f.name* in Figure 29 and Figure 31). The referencing feature then defines a start point on its UCM with an implicit precondition that matches the postcondition from the referenced feature exit. Thus, the scenario will continue from the referencing start point when the referenced feature exit is reached. This is achieved by labeling the end and start points the same (preferably with the feature exit name) and placing a guard right after the start point. The guard shows the actual precondition in square brackets (see *[recall_timeout]* and *[ACD]* in Figure 37) and allows multiplexing multiple scenarios caused by different events/preconditions onto one path.

Similarly, all *name* start points (see Figure 29 and Figure 31) define implicit preconditions. The referencing feature then defines an end point on its UCM with an implicit postcondition that matches the precondition of the *name* start point. Thus, the scenario will continue from the referenced start point if the referencing end point is reached. Once again this is achieved by labeling the end and start points the same (preferably with the name of the start point).

Sometimes, a referencing feature necessitates the insertion of an *end point/start point pair* into the map of the referenced feature. This pair consists of an end point connected to a labeled start point (see *recall* in Figure 32). The pair allows the definition of an entry point into an existing feature. This entry point can be used by another feature to continue a scenario with an existing feature. The same referencing mechanism is used for start points in end point/start point pairs as is used for start points in event stub or FI fork structures.

6.5 Resulting Context

Figure 28 shows how the solution balances the forces identified in 3.1.3. All in all, the "Isolation and Integration" UCM style balances well the evolveability force at the expense of the simplicity of the UCMs and thus tool independence. The evolveability force is sufficiently balanced because:

- Features are not polluted since the hooks required by other features can be clearly identified. A reader of the feature maps interested only in the feature itself can simply ignore all feature exits of event stubs, all FI forks, and all end points connected to start points. Since all features interacting with the current feature are multiplexed onto the same feature exit

paths, variations of behavior introduced by the interacting features do not require additional paths to be shown on the current feature's map (as it was the case with the "Standard Root Map" UCM style).

- Feature maps are not distributed since all feature maps are connected to and can be accessed from the feature's own root map.
- The scaleability force is well balanced because the pollution and distribution problems have been addressed.
- Feature interaction detection is as possible as with the "Standard Root Map" UCM style since features are integrated with each other. The specification of feature interactions, however, does not lead to feature pollution thus slightly improving the balance of the feature interaction force. The improvement, however, is not enough to affect the force hierarchy.
- Because consistency is enforced by the integration of features, the reuseability force remains relatively well balanced. The force, however, is not as well balanced as compared to the "Standard Root Map" UCM style since more than one reusable unit may exist on a single UCM.
- The overall understandability of the feature has been improved significantly from the "Standard Root Map" UCM style since features are now not polluted, feature maps are now not distributed, the total number of maps decreases compared to the "Standard Root Map" UCM style, and the number of levels per feature is kept low since a general root map is not introduced.

Although the "Isolation and Integration" UCM style balances well the evolveability force, this approach is more dependent on tool support than others because of the complexity of the maps and the frequent use of stubs. Furthermore, some problems regarding the navigability of UCMs and the degree of completeness of feature descriptions have not been addressed.

The complexity of the maps has two reasons. First, the event stub, FI fork, and end point/start point pair structures introduce a certain amount of complexity due to the number of required stubs. Second, the integration of features results in a somewhat less intuitive definition of a feature than the "Individual Maps" UCM style. Paths may be disjoint making it difficult to follow the causal flow (e.g. see Figure 37). Furthermore, no visual distinction between referencing start and end points and non-referencing start and end points is given, thus adding further complexity. In general, a navigation problem exists since the references between UCMs of different features are implicit and cannot be traversed by the tool and since the main start point of a feature is not apparent. The pattern "Explicit Navigation" in section 7 addresses these issues.

Moreover, the definitions of features without the referenced maps may not be as complete as one would want. For instance, the ACD feature in Figure 37 does not give you a clue of what happens after the *s2.t* end point. The reader of the UCMs has to look at the referenced map (the Basic Call root map) to find out. Therefore, a feature root map by itself does not quite define complete scenarios, rather only the extensions and variations. The patterns "Explicit Navigation" in section 7 and "Explicit Causality of Highest-Level Interaction" in section 8 address this problem.

Another issue is that user actions identified in Figure 7 and Figure 8 of the "Individual Maps" UCM style and in Figure 15 of the "Standard Root Map" UCM style (such as *start call*, *directory number*, *answer*, and *end call*) have been lost by the "Isolation and Integra-

tion" UCM style. In addition to the lost naming, the causal relationship between user actions is still not shown. This leads to a more complicated map because it is more difficult to understand the feature. The patterns "Explicit Causality of Highest-Level Interaction" in section 8 and "Context View of Feature" in section 9 tackle these issues.

6.6 Known Uses

Further examples of call control features that make use of the event stub structure in addition to the ones shown in Section 6.7 are Hold and Conference.

6.7 Examples of "Isolation and Integration" UCM Style

Examples of two features documented using the "Isolation and Integration" UCM style are shown in Figure 32 to Figure 36 (Basic Call) and Figure 37 (Automatic Call Distribution).

6.7.1 Basic Call

The Basic Call root map (Figure 32) and its four plug-in maps (Dialing, Wait For Answer, Wait For Answer.t, Talking) specify exactly the same behavior as explained for the "Individual Maps" style (see 4.7.1). To understand Basic Call, one can simply ignore all feature exits, all FI forks, and all end points connected to labeled start points. These structures provide only the hooks for other features to alter Basic Call behavior.

Once again, in and out labels specify the bindings between stubs on the Basic Call map and their plug-in maps. The naming convention for

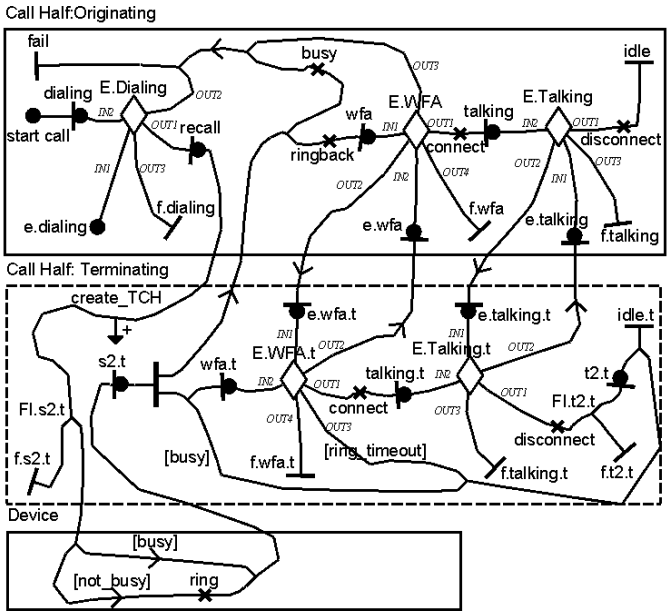


Figure 32: Basic Call root map (Isolation and Integration)

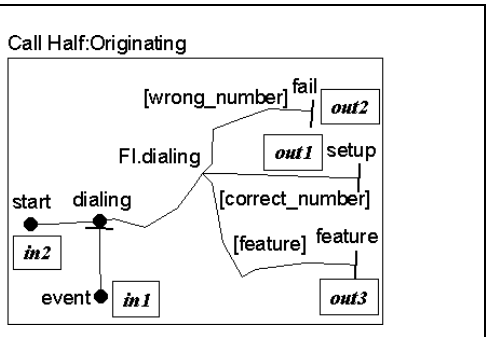


Figure 33: E.Dialing
(Isolation and Integration – Basic Call)

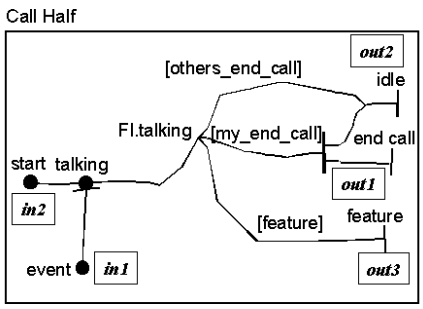


Figure 34: E.Talking
(Isolation and Integration – Basic Call)

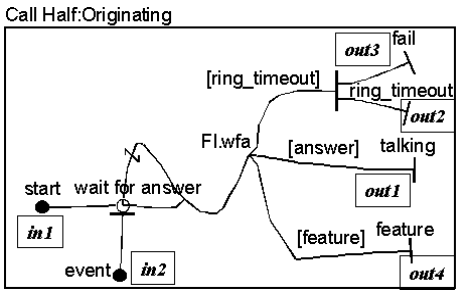


Figure 35: E.WFA
(Isolation and Integration – Basic Call)

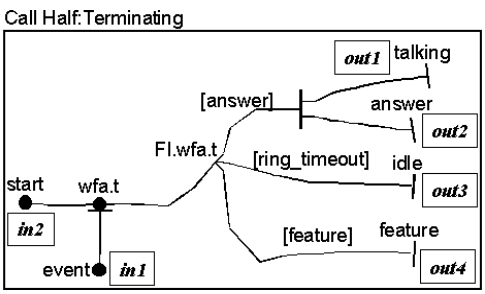


Figure 36: E.WFA.t
(Isolation and Integration – Basic Call)

event stubs is illustrated in Figure 29. The following list defines which stubs in Figure 32 contain which plug-in maps:

- the *E.Dialing* stub contains the *E.Dialing* plug-in (see Figure 33),
- the *E.Talking* and *E.Talking.t* stubs contain the *E.Talking* plug-in (see Figure 34),
- the *E.WFA* stub contains the *E.WFA* plug-in (see Figure 35), and
- the *E.WFA.t* stub contains the *E.WFA.t* plug-in (see Figure 36).

The event stub structure was applied five times to the Basic Call feature (*E.Dialing*, *E.WFA*, *E.WFA.t*, *E.Talking*, and *E.Talking.t*). The FI fork structure was applied twice (*FI.s2.t* and *FI.t2.t*) and the end point connected to a labeled start point was applied once (*recall*) as required by ACD. Note that the dynamic stubs from the "Standard Root Map" UCM style correspond to the FI forks of the "Isolation and Integration" UCM style. The following list shows the definitions of all FI forks of the Basic Call feature:

- *FI.s2.t* and *FI.t2.t* (both see Figure 32):
  IF (ACD) THEN feature
- *FI.dialing* (see Figure 33):
  IF ((NOT wrong_number) AND (NOT correct_number)) THEN feature
- *FI.talking* (see Figure 34):
  IF ((NOT my_end_call) AND (NOT others_end_call)) THEN feature
- *FI.wfa* (see Figure 35) and *FI.wfa.t* (see Figure 36):
  IF ((NOT ring_timeout) AND (NOT answer)) THEN feature

6.7.2 Automatic Call Distribution (ACD)

The new ACD root map (Figure 37) references the Basic Call root map to specify the same behavior as explained in the "Individual Maps" style (see 4.7.2). The following list defines which start and end points on the ACD root map align with which

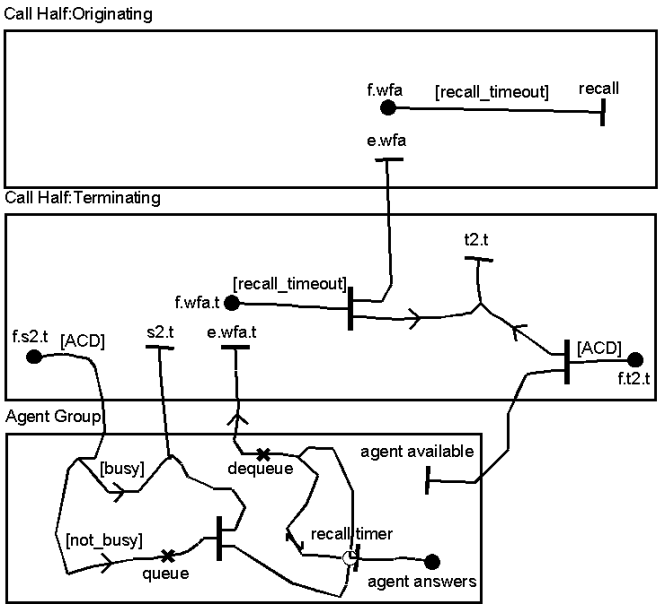


Figure 37: ACD root map (Isolation and Integration)

end and start points on the Basic Call root map:

- the *f.s2.t*, *f.wfa*, *f.wfa.t*, and *f.t2.t* start points on the ACD root map references the end points with the same name on the Basic Call root map,
- the *e.wfa*, *e.wfa.t*, *recall*, *s2.t*, and *t2.t* end points references the start points with the same name.

The following guards have been added to the feature description as required by the integration step:

- the *[ACD]* guards after the *f.s2.t* and *f.t2.t* start points, and
- the *[recall_timeout]* guards after the *f.wfa.t* and *f.wfa* start points.

Therefore, the ACD feature starts at the *f.s2.t* start point once the *f.s2.t* end point on the Basic Call root map is reached. After performing ACD specific actions, the scenario continues in Basic Call at start point *s2.t*. If the agent *answers* or the *recall timer* expires, the scenario reaches the *e.wfa.t* end point and therefore continues from the *e.wfa.t* start point in Basic Call. From the naming conventions, one can deduce that the *answer* scenario requires normal Basic Call behavior whereas the *recall timeout* scenario requires new behavior as shown on the ACD root map. Note how the path continues from *e.wfa.t* via the Basic Call root map to *f.wfa.t* if the *recall timeout* scenario occurs. The recall timeout scenario ends at *t2.t* and *e.wfa*. on the ACD root map. Therefore, it continues at the *t2.t* start point on the Basic Call root map and at the *f.wfa* start point (similarly to *e.wfa.t* and *f.wfa.t*) on the ACD root map. Finally, it ends at the *recall* end point which continues at the *recall* start point on the Basic Call root map.

7 Explicit Navigation

Disjoint paths and implicit references make it difficult to follow causal flow.
Therefore:
Use pre/postcondition stubs to join paths, navigate explicitly from map to map, and show complete scenarios.

8 Explicit Causality of Highest-Level Interaction

Ambiguous feature descriptions are often caused by implicit causal relationships of highest-level user interactions with the system.
Therefore:
Use paths with “at-location” markers to explicitly show highest-level causal relationships.

9 Context View of Feature

Explicit highest-level causal relationships on a single map and lost naming of user actions make UCMs unnecessarily complex.
Therefore:
Use higher level UCMs for clear “Context View” of features.

10 Conclusion

The characteristics of a system determine the most effective UCM style. If a few key features need to be captured rapidly and independently, the "Individual Maps" UCM style is most useful. If a small but evolving system consisting of interacting features needs to be specified, the "Standard Root Map" UCM style is most appropriate. If a large, evolving system with many interacting features needs to be documented, the "Isolation and Integration" UCM style is best applied. This paper introduces patterns for each UCM style, each pattern providing guidelines on how and when to use the style.

Future work includes the extension of this pattern collection with a more detailed description of the three patlets and four additional patterns. Furthermore, the existing patterns need to be reevaluated regularly since changes to the UCM Navigator tool may impact these patterns. Finally, the patterns themselves may impact or spur the development of new features for the UCM Navigator tool and other such tools.

11 References

1. Amyot, D., *Use Case Maps as a Feature Description Language*, in S. Gilmore and M. Ryan (Eds), *Language Constructs for Designing Features*, pp. 27-44, Springer-Verlag, 2000.
2. Amyot, D. and Andrade, R., *Description of Wireless Intelligent Network Services with Use Case Maps*, in *SBRC'99, 17th Brazilian Symposium on Computer Networks*, Salvador, Brazil, May 1999.
3. Amyot, D. and Logrippo, L., *Use Case Maps and LOTOS for the Prototyping and Validation of a Mobile Group Call System*, in *Computer Communication*, 23(8), April 2000.
4. Amyot, D., Buhr, R.J.A., Gray, T., and Logrippo, L., *Use Case Maps for the Capture and Validation of Distributed Systems Requirements*, in *RE'99, Fourth IEEE International Symposium on Requirements Engineering*, pp. 44-53, Limerick, Ireland, June 1999.
5. Amyot, D., Logrippo, L., and Buhr, R.J.A., *Spécification et conception de systèmes communicants : une approche rigoureuse basée sur des scénarios d'usage*, in *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP'97)*, pp. 159-174, Hermes, Paris, 1997.
6. Andrade, R., *Applying Use Case Maps and Formal Methods to the Development of Wireless Mobile ATM Networks*, in *Lfm2000: The Fifth NASA Langley Formal Methods Workshop*, Williamsburg, Virginia, USA, June 2000.
7. Andrade, R. and Logrippo, L., *Reusability at the Early Development Stages of the Mobile Wireless Communication Systems*, in *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2000), Vol. VII, Computer Science and Engineering: Part I*, pp. 11-16, Orlando, Florida, July 2000.
8. Buhr, R.J.A., *Use Case Maps as Architectural Entities for Complex Systems*, in *Transactions on Software Engineering*, pp. 1131-1155, IEEE, December 1998.
9. Buhr, R.J.A. and Casselman, R.S., *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, USA, 1995.
10. Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., and Mankovski, S., *High Level, Multi-agent Prototypes from a Scenario-Path Notation: A Feature-Interaction Ex-*

ample, in *PAAM'98, 3rd Conference on Practical Application of Intelligent Agents and Multi-Agents*, London, UK, March 1998.
11. Cameron, D. et al., *Draft Specification of the User Requirements Notation*, Canadian contribution CAN COM 10-12 to ITU-T, November 2000.
12. Chung, L., Nixon, B.A., Yu, E., and Mylopoulos, J., *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, 2000.
13. Elammari, M. and Lalonde, W., *An Agent-Oriented Methodology: High-Level and Intermediate Models*, in *Proceedings of the 1st International Workshop on Agent-Oriented Information Systems (AOIS'99)*, Heidelberg, Germany, June 1999.
14. *Goal-oriented Requirement Language (GRL) Web Site*, 2001, http://www.cs.toronto.edu/km/GRL/.
15. Gross, D. and Yu, E., *From Non-Functional Requirements to Design through Patterns*, in *Requirements Engineering*, 6:18-36, Springer-Verlag, 2001.
16. Miga, A., *Application of Use Case Maps to System Design with Tool Support*, M.Eng. thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, October 1998, http://www.UseCaseMaps.org/tools/ucmnav/.
17. Miga, A., Amyot, D., Bordeleau, F., Cameron, D., and Woodside, M., *Deriving Message Sequence Charts from Use Case Maps Scenario Specifications*, 10th SDL Forum, Copenhagen, Denmark, June 2001.
18. Nakamura, M., Kikuno, T., Hassine, J., and Logrippo L., *Feature Interaction Filtering with Use Case Maps at Requirements Stage*, in *Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00)*, Glasgow, Scotland, UK, May 2000.
19. Sales, I. and Probert, R., *From High-Level Behaviour to High-Level Design: Use Case Maps to Specification and Description Language*, in *SBRC 2000, 18th Brazilian Symposium on Computer Networks*, Belo Horizonte, Brazil, May 2000.
20. Scratchley, W.C. and Woodside, C.M., *Evaluating Concurrency Options in Software Specifications*, in *MASCOTS'99, Seventh International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 330-338, College Park, MD, USA, October 1999.
21. *Use Case Maps Web Site* and *UCM User Group*, 1999, http://www.UseCaseMaps.org.
22. Weiss, M., *Patterns and Non-Functional Requirements*, Presentation at CITO Software Research Review, March 2001, http://fusion.scs.carleton.ca/~weiss/research/nfr/cito.pdf.
23. Zave, P., *Requirements for evolving systems: A telecommunications perspective*, in *RE'01, Fifth IEEE International Symposium on Requirements Engineering*, pp. 2-9, Toronto, Canada, August 2001, http://www.research.att.com/~pamela/re1.pdf.